Manual
PASM

Lifeboat Associates
1651 Third Avenue
New York, N.Y. 10028
Tel: (212) 860-0300
TWX: 710-581-2524 (LBSOFT NYK)
Telex: 640693 (LBSOFT NYK)

PHOENIX SOFTWARE ASSOCIATES LTD.

PASM

User´s Manual

Revision 1.0

February 1, 1980

Written by Neil J. Colvin

LIFEBOAT ASSOCIATES
1651 Third Avenue
New York, N. Y. 10028
Tel: (212) 860-0300
TWX: 710-581-2524
Telex: 640693

Chapter 1

Introduction

PASM is the Pheonix Software Associates Ltd. symbolic assembly program for microprocessors which execute the Z80 (a trademark of Zilog, Inc.) instruction set. It is a two-pass assembler (requiring the source program to be read twice to complete the assembly process) designed to run under the Phoenix Software Associates Ltd. PDOS (or similar) operating system. It is therefore device independent, allowing complete user flexibility in the selection of standard input and output device options.

The assembler performs many functions, making machine language programming easier, faster, and more efficient. Basically, the assembler processes the Z80 programmer's source program statements by translating mnemonic operation codes to the binary codes needed in machine instructions, relating symbols to numeric values, assigning relocatable or absolute memory addresses for program instructions and data, and preparing an output listing of the program which includes any errors encountered during the assembly.

The PSA Macro Assembler also contains a powerful macro capability which allows the programmer to create new language elements, thus expanding and adapting the assembler to perform specialized functions for each programming job.

In addition, the PSA Assembler provides the facilities required to specify program module linkages, allowing the PSA Linkage Editor (LINK) to link independently assembled program modules together into a single executable program. This allows for the modular and systematic development of large programs, and for easy sharing of common program modules among different programs.

Statements

Assembler programs are usually prepared on a terminal, with the aid of a text editing program. A program consists of a sequence of statements in the assembly language. Each statement is normally written on one line, and terminated by a carriage return/line feed sequence. PSA Macro Assembler statements are free-format. This means that the various statement elements are not placed at a specific numbered column on the line.

There are four elements in an assembler statement (three of which are optional), separated from each other by specific characters. These elements are identified by their order of appearance in the statement, and by the separating (or delimiting) character which follows or precedes the elements.

Statements are written in the general form:

```
label: operator   operand,operand ;comment <CR-LF>
```

The assembler converts statements written in this form into the binary machine instructions.

Instruction Formats

The Z80 uses a variable length instruction format. A given machine instruction may be one, two, three, or four bytes long depending on the specific machine code and on the addressing mode specified. The PSA Assembler automatically produces the correct number of machine code bytes for the particular operation specified. Appendix A specifies the various machine code mnemonics accepted by the assembler and the format of the operands required.

Statement Format

As previously described, assembler statements consist of a combination of a label, an operator, one or more operands, and a comment; the particular combination depends on the statement usage and operator requirements.

The assembler interprets and processes these statements, generating one or more binary instructions or data bytes, or performing some assembly control process. A statement must contain at least one of these elements, and may contain all four. Some statements have no operands, while others may have many.

Statement labels, operators, and operands may be represented numerically or symbolically. The assembler interprets all symbols and replaces them with a numeric (binary) value.

Symbols

The programmer may create symbols to use as statement labels, as operators, and as operands. A symbol may consist of any combination of from one to six characters from the following set:

The 26 letters: A-Z
Ten digits: 0-9
Three special characters:
- $ (Dollar Sign)
- % (Percent)
- . (Period)

These characters constitute the Radix-40 character set (so named because it contains only 40 characters). Any statement character which is not in the Radix-40 set is treated as a symbol delimiter when encountered by the assembler.

The first character of a symbol must not be numeric. Symbols may also not contain embedded spaces. A symbol may contain more than six characters, but only the first six are used by the assembler.

The PSA Assembler will accept programs written using both upper and lower case letters and symbols. Lower case letters are treated as upper case in symbols. Additional special characters and lower case letters elsewhere are taken unchanged.

Labels

A label is the symbolic name created by the programmer to identify a statement. If present, the label is written as the first item in a statement, and is terminated by a colon (:). A statement may contain more than one label, in which case all identify the same statement. Each label must be followed by a colon, however. A statement may consist of just a label (or labels), in which case the label(s) identifies the following statement.

When a symbol is used as a label, it specifies a symbolic address. Such symbols are said to be defined (have a value). A defined symbol can reference an instruction or data byte at any point in the program.

A label can be defined with only one value. If an attempt is made to redefine a label with a different value, the second value is ignored, and an error is indicated.

The following are legal labels:

```
$SUM:
ABC:
B123:
WHERE%:
```

The following are illegal:

```
30QRT:     (First character must not be a digit)
AB CD:     (Cannot contain embedded space)
```

If too many characters are used in a label, only the first six are used. For example the label ZYXWVUTSR: is recognized by the assembler to be the same as ZYXWVUABC:.

Operators

An operator may be one of the mnemonic machine instruction codes, a pseudo-operation code which directs the assembly process, or a user defined code (either pseudo-op or macro). The assembler pseudo-op codes are described in Chapter 3 and summarized in Appendix B.

The operator element of a statement is terminated by any character not in the Radix-40 set (usually a space or a tab). If a statement has no label, the operator must appear first in the statement.

A symbol used as an operator must be predefined by the assembler or the programmer before its first appearance as an operator in a statement.

Operands

Operands are usually the symbolic addresses of the data to be accessed when an instruction is executed, the names of processor registers to be used in the operation, or the input data or arguments to a pseudo-op or macro instruction. In each case, the precise interpretation of the operand(s) is dependent on the specific statement operator being processed. Operands are separated by commas, and are terminated by a semicolon (;) or a carriage return/line feed.

Symbols used as operands must have a valued predefined by the assembler or defined by the programmer. These may be symbolic references to previously defined labels where the arguments used by this instruction are to be found, or the symbols may represent constant values or character strings.

Comments

The programmer may add a comment to a statement by preceding it with a semicolon (;). Comments are ignored by the assembler but are useful for documentation and later program debugging. The comment is terminated by the carriage return/line feed at the end of the statement. In certain cases (eg. conditional assembly and macro definitions), the use of the left and right square brackets ([]) should be avoided in a comment as it could affect the assembly process.

An assembler statement may consist of just a comment, but each such statement must begin with a semicolon.

Statement Processing

The assembler maintains several internal symbol tables for recording the names and values of symbols used during the assembly. These tables are:

1. Macro Table - This table contains all macros. It is initially empty, and grows as the programmer defines macros.

2. Op-Code Table - This table contains all of the machine operation mnemonics (op-codes), the assembler pseudo-operations (pseudo-ops), and user defined operators (.OPSYNs). It initially contains the basic op-codes and pseudo-ops, and grows as the programmer provides additional definitions.

3. Symbol Table - This table contains all programmer-defined symbols other than those described above. It initially contains the standard register names, and then grows as new symbols are defined.

Internally, all of these tables occupy the same space, so that all of the available space can be used as required.

Order of Symbol Evaluation

The following table shows the order in which the assembler searches the tables for a symbol appearing in each of the statement fields:

Label Field:

1. Symbol followed by a colon. If no colon is found, no label is present.

Operator Field:

1. Macro
2. Machine operator
3. Assembler operator
4. Symbol

Operand Field:

1. Number
2. Macro
3. Symbol
4. Machine operator

Because of the different table searching orders for each field, the same symbol could be used as a label, an operator, and a macro, with no ambiguity.

Programmer-Defined Symbols

There are two type of programmer-defined symbols: labels and assignments. As previously described, labels are generated by entering a symbol followed by a colon (e.g. LABEL:). Symbols used as labels cannot be redefined with a different value once they have been defined. The value of a label is the value of the location counter at the time the label is defined.

Assignments are used to represent, symbolically, numbers, bit patterns, or character strings. Assignments simplify the program development task by allowing a single source program modification (the assignment statement) to change all uses of that number or bit pattern throughout the program. Symbols given values in an assignment statement may have new values assigned in subsequent statements. The current value of an assigned symbol is the last one given to it.

A symbol may be entered into the symbol table with its assigned value by using a direct assignment statement of the form:

```
symbol = value {; or CR-LF}
```

where the value may be any valid numeric value or expression.

The value assigned to the symbol may subsequently be changed by another direct asignment statement.

The following are valid assignment statements:

```
VALUE1 = 23
SIZE = 4*36
ZETA = SIZE
```

If it is desired to fix the value assigned to a symbol so that it cannot subsequently be redefined, the direct fixed assignment statement should be used. This statement is the same as the direct assignment statement except that the symbol is followed by two equal signs instead of one. For example:

```
FIXED == 46
NEWVAL == SIZE
```

Assembly-Time Assignments

It is often desirable to defer the assignment of a value to a symbol until the assembly is actually underway (i.e. not specify the value as part of the source program). This is especially useful in setting program origin, buffer sizes, and in specifying parameter values which will be used to control conditional assembly pseudo-ops.

The PSA Assembler provides the ability to specify symbols with values to be determined at assembly time, and the mechanism by which the values may be interactively defined. To specify an assembly-time assignment, the following format is used:

```
symbol =\ {svalue}
```

where the svalue in braces indicates the optional specification of a message to be output on the console device at assembly time before requesting the symbol's value. Any valid string value may be specified, including multi-line values.

After the optional message is output on the console, a colon (:) is output to indicate that the assembler is waiting for the desired value to be entered. The value which is to be assigned to the symbol is then input on the console device and the assembly continues with the symbol having the specified value. This interaction only occurs during the first assembly pass. The symbol's value remains unchanged during subsequent passes.

Only numeric values may be entered through the console in this fashion. The number which is input must conform to the same rules as any other number used in the assembly source program, and may be followed by an optional radix modifier (see the section on Numbers below). The number is assumed to be decimal unless followed by a radix modifier.

The value being input is not processed until a carriage return is entered. Any mistyped character may be deleted by the use of the DELETE (or RUBOUT) key (which will echo the deleted character) or the BACKSPACE (or CTL-H) key which will backspace over the deleted character, and the entire number may be deleted by entering CTL-U (simultaneous use of the CTRL and the U key). Any character which is input but is not valid as part of a number will be echoed as a BELL and will be ignored.

The following are examples of assembly-time assignment statements:

```
BUFSIZ =\ "BUFFER SIZE (50 TO 500 CHARACTERS)"
DISK =\ "VERSION (0-PAPER TAPE 1-DISK)"
```

Assembly-time assignment statements are similar to direct fixed assignments (==) in not allowing the symbol to be redefined elsewhere in the program.

Local and Global Symbols

When assembling a large program, it is sometimes difficult to keep track of the symbols used for local data references and branching. To facilitate modular programming, the PSA Assembler provides for both global and local symbols within a single program. All symbols which start with two periods are defined as being local, and all other symbols are global. For example, the following are valid local symbols:

```
..ABCD:
..1234:
..:
```

A particular occurrence of a local symbol is only defined within the boundaries of its enclosing global symbols. For example, in the following sequence of label definitions, the symbol ..SYM1 is only defined (and can only be referenced) within the program between the definition of GLOB1 and GLOB2:

```
      ...
GLOB1:
      ...
..SYM1:
      ...
GLOB2:
      ...
```

This localization of symbol definitions allows the same symbol to be used unambiguously more than once in the program. It also simplifies program understandability by immediately differentiating between local and global symbols.

In addition to labels, any other programmer-defined symbol may be specified as local (e.g. macros) in the same manner. Because of the local usage of these symbols, they do not appear in the symbol table listing or in the symbol table optionally output to the object file.

External, Internal, and Entry Symbols

Programmer-defined symbols may also be used as external, internal, and entry point symbols in addition to their appearance as labels or in assignment statements.

Symbols which fall into one of these three groups are different from other symbols in the program because they can be referenced by other, separately assembled, program modules. The manner in which they are used depends on where they are located: in the program in which they are defined, or in the program in which they are a reference to a symbol defined elsewhere.

If the symbol appears in a program in which it is defined, it must be declared as being available to other programs by the use of the pseudo-ops .INTERN or .ENTRY, or through the use of the delimiters "::", "=:", "==:", or "=\:" in their definition statements. These special delimiters are exactly equivalent to the sequence:

```
.INTERN symbol
symbol <delimiter without colon (:)>
```

In each case, the delimiter is the normal symbol definition operator (:, =, ==, =\) with an additional colon (:) added to indicate an internal symbol definition.

If the symbol is located in a program in which it is a reference to a symbol defined in another program, it must be declared as external by the use of the .EXTERN pseudo-op, or through the use of the "#" symbol modifier. This special symbol modifier is appended to the end of any symbol to declare it external. For example, the statement:

```
LXI H,SYMBOL#
```

is exactly equivalent to:

```
.EXTERNAL SYMBOL
LXI H,SYMBOL
```

Numbers

Numbers used in a program are interpreted by the assembler according to a radix (number base) specified by the programmer, where the radix may be 2 (binary), 8 (octal), 10 (decimal), or 16 (hexadecimal). The programmer uses the .RADIX pseudo-op to set the radix for all numbers which follow. If the .RADIX statement is not used, the assembler assumes a radix of 10 (decimal).

The radix may be changed for a single number by appending a radix modifier to the end of the number. These modifiers are B for binary, O or Q for octal, D or . (period) for decimal, and H for hexadecimal. To specify the hexadecimal digits, the letters A through F are used for the values 10 through 15 decimal. All numbers, however, must begin with a numeral. For example, the following are valid numbers:

10	10 in current radix
10.	10 decimal
10B	10 binary (2 decimal)
0FFH	FF hexadecimal (255 decimal)

The following are invalid numbers:

14B	4 is not a binary digit
FFH	the number must start with a numeral

Arithmetic and Logical Operations

Numbers and defined symbols may be combined using arithmetic and logical operators. The following operators are available:

+	Add (or unary plus)
-	Subtract (or unary minus)
*	Multiply
/	Integer division (remainder discarded)
@	Integer remainder (quotient discarded)
&	Logical AND
!	Logical inclusive OR

^   Logical exclusive OR (or unary radix change)
#   Logical unary NOT
<   Left binary shift
>   Right binary shift

The assembler computes the 16-bit value of a series of numbers and defined symbols connected by these operators. All results are truncated to the left, if necessary. Two's complement arithmetic is used, with the meaning of the sign bit (the most significant bit) being left to the programmer. This means that a numeric value may be either between 0 and 65,535 or between -32,768 and 32,767, depending on whether it is signed or unsigned.

These combinations of number and defined symbols with arithmetic and logical operators are called expressions. When evaluating an expression, the assembler performs the specified operations in a particular order, as follows:

1. Unary minus or plus (- +)
2. Unary radix change (^B ^O ^Q ^D ^H)
3. Left and right binary shift (< >)
4. Logical operators (& ! ^ #)
5. Multiply/Divide (* /)
6. Remainder (@)
7. Add/Subtract (+ -)

Within each of the above groups, the operations are performed from left to right. For example, in the expression:

```
-ALPHA+3*BETA/DELTA&^H55
```

the unary minus of ALPHA is done first, then DELTA is ANDed with a hexadecimal 55, then BETA is multiplied by 3, the result of which is divided by the result of the AND, and finally, that result is added to the negated ALPHA.

To change the order in which the operations are performed, parentheses may be used to delimit expressions and to specify the desired order of computation. Each expression within parentheses is considered to be a single numeric value, and is completely evaluated before it is used to compute any further values. For example, in the expression:

```
4*(ALPHA+BETA)
```

the addition of ALPHA to BETA is peformed before the multiplication

Radix Change Operator

The radix change operator is used to temporarily change the radix in which a following number or expression is to be interpreted. It is written as an up-arrow (^) followed by the radix modifier of the desired radix. These modifiers are the same as those used to specify the radix of a single number (B-binary, O or Q-octal, D-decimal, and H-hexadecimal). The radix change only affects the immediately following number or parenthesized numeric expression. For example, all of the following are valid representations of the decimal number 33:

```
33.
33D
^D33
^D(10*3+3)
^D(10*THREE+THREE)
^D10*^D3+^D3
```

but the following is not a representation of decimal 33 if the prevailing radix is not decimal:

```
^D3*10+3
```

because the radix change only affects the value immediately following it, in this case 3.

Binary Shifting

The binary shift operators (< left, > right) are used to logically shift a 16-bit value to the left or right. The number of places to be shifted is specified by the value following the shift operator. If that value is negative, the direction of the shift is reversed. For example, all of the following expressions have a value of 4 decimal:

```
8>1
1<2
2>-1
```

One-byte Values

All of the above discussion has been based on the computation of 16-bit (two byte) numeric values. Many of the Z80 operations require an 8-bit (one byte) value. Since all computations are done as a 16-bit value, an operation calling for only eight bits will discard the high order eight bits (the most significant byte) of the value. If the byte discarded is not either zero or minus one (all one bits), a warning will be given on the assembly listing.

Character Values

To generate a binary value equivalent to the ASCII representation of a character string, the single (') or double (") quotation mark is used. The character string is enclosed in a pair of the quotation marks. For example, all of the following are valid character values:

```
"A"
'B'
"AB"
'CD'
```

Note that whichever quotation mark is used to initiate the character string must be used to terminate it. If the string is longer than two bytes, it is truncated to the left. Each 7-bit ASCII character is stored in an 8-bit byte, with the high-order bit set to zero.

A character string of this type may be used wherever a numeric value is allowed.

A single quote may be used inside a string delimited by double quotes, and vice-versa. If it is necessary to use a single quote within a string delimited by single quotes, two single quotes must be used. The same is true for a double quote in a string delimited by double quotes.

Location Counter Reference

The location counter may be referenced as a numeric 16-bit value by the use of the symbol . (period). The value represented by . is always the location counter value at the start of the current assembly language statement. For example:

```
JMP .
```

is an effective error trap, jumping to itself continuously.

Temporary Variables

In addition to named symbols, it is sometimes convenient to be able to reference assembly time variables by a numeric value or subscript. The PSA Macro Assembler allows the definition of a single global assembly time array for this purpose. This array has the special name "*" and is referenced by "*[subscript]" where subscript is any valid 16-bit absolute expression with a value between 0 and the number of array elements minus 1. This array reference may occur anywhere a normal symbol or label reference or definition is allowed. A temporary variable may be assigned any numeric value, either absolute or relocatable, within the normal 16-bit restriction. The allocation of space for this array is done by the .TEMPS pseudo-op described later.

For example:

```
V=5
*[V]=4*V
W=5
MVI A,*[W]+23
```

String Variables

In addition to being able to assign a numeric value to a symbol, the PSA Macro Assembler allows a symbol to be assigned a string value as well. This is done through the .DEFINE pseudo-op in the same manner in which a macro is defined (see Chapter 4), except that defined string variables may have no arguments. A string variable may only be DEFINEd once in a program. String variables may be used anywhere a string value is called for (eg. the .ASCII pseudo-op).

For example:

```
.DEFINE PROMPT=[>>]
.ASCII "ENTER NUMBER",PROMPT
```

Substring Operator

It is sometime desirable to obtain a substring of a given string, macro argument or string variable. The PSA Macro Assembler provides that capability through the use of the substring operator "<". The format for the use of this operator is:

```
<[s{,l}]dtextd
```

or

```
<[s{,l}]svariable
```

where s is a 16-bit expression whose value is the starting character to be used, and l is an optional 16-bit expression for the length of the string to use. The first string character is 0. If the length is omitted, the entire rest of the string is used.

The first form uses a delimited string as its string argument. the d represents a delimiter which may be any character not contained within the string itself. The string argument starts with the first character after the delimiter, and terminates with the last character before the corresponding delimiter. The delimiters are not part of the string. If the initial delimiter is a left bracket ([), then the matching delimiter is a right bracket (]). The left and right brackets are paired, so that intervening pairs of left and right brackets will not terminate the string.

The second form uses a string variable as its string argument.

For example:

```
.DEFINE STRNG1=[THIS IS STRING ONE]
.ASCII   <[0,1]STRNG1
.ASCII <[5,3]"0123456789"
```

The substring operator may also be used in a normal arithmetic expression in place of a single or double character value. For example:

```
MVI A,<[V,1]"0123456789"
```

String Length Operator

It is often quite necessary to use the length of a given string value as a value in an numeric expression. To facilitate this, the string length operator "@" is provided. It is used by prefixing it to the front of the string value whose length is desired, and may be used anywhere an absolute number is allowed. For example:

```
.DEFINE STRNG1=[THIS IS A STRING]
MESS1: .ASCIZ STRNG1
LXI H,MESS1
MVI B,@STRNG1
```

String Values

Many pseudo-ops utilize a generic "string value" argument. A string value is either a delimited string (see Substring Operator above), a string variable or a substring operation. For example, all of the following are valid string values (assuming STRNG1 as defined above):

```
STRNG1
"THIS IS A STRING"
[THIS IS A [BRACKETED] DELIMITED STRING]
<[0,5]"THIS IS A SUBSTRING"
<[3,2]STRNG1
```

Chapter 2

Addressing and Relocation

Address Assignment

As source statements are processed by the assembler, consecutive memory addresses are assigned to the instruction and data bytes of the object program. This is done by incrementing an internal program counter each time a memory byte is assigned. Some statements may increment this internal counter by only one, while others could increase it by a large amount. Certain pseudo-ops and direct assignment statements have no effect on the counter at all.

In the program listing generated by the assembler, the address assigned to every statement is shown.

Relocation

The PSA Macro Assembler will create a relocatable object program. This program may be loaded into any part of memory as a function of what has been previously loaded. To accomplish this, certain 16-bit values which represent addresses within the program must have a relocation constant added to them. This relocation constant, added when the program is loaded into memory, is the difference between the memory location an instruction (or piece of data) is actually loaded into, and the location it was assembled at. If an instruction had been assembled at location 100 (decimal), and was loaded into location 1100 (decimal), then the relocation constant would be 1000 (decimal).

Not all 16-bit quantities must be modified by the relocation constant. For example, the instruction:

```
LXI H,00FFH
```

references a 16-bit quantity (00FFH) which does not need relocation. However, the set of instructions:

```
JZ DONE
   .
   .
   .
DONE:
```

does reference a 16-bit quantity (the address of DONE) which must be relocated, since the physical location of DONE changes depending on where the program is loaded into memory.

To accomplish this relocation, the 16-bit value forming an address reference is marked by the assembler for later modification by the loader or linkage editor. Whether a particular 16-bit value is so marked depends on the evaluation of the arithmetic expression from which it is obtained. A constant value (integer) is absolute (not relocatable), and never modified. Point references (.) are relocatable (assuming relocatable code is being generated), and are always modified by the loader or linkage editor. Symbolic references may be either absolute or relocatable.

If a symbol is defined by a direct assignment statement, it may be absolute or relocatable depending on the expression following the equal sign (=). If the symbol is a label (and relocatable code is being generated) then it is relocatable.

To evaluate the relocatability of an expression, consider what happens at load or linkage edit time. A relocation constant, r, must be added to each relocatable element, and the expression evaluated. For example, in the expression:

```
Z = Y+2*X-3*W+V
```

where V, W, X, and Y are relocatable. Assume that r is the relocation constant. Adding this constant to each relocatable term, the expression becomes:

```
Z(r) = (Y+r)+2*(X+r)-3*(W+r)+(V+r)
```

By rearranging the expression, the following is obtained:

```
Z(r) = Y+2*X-3*W+V + r
```

This expression is suitable for relocation because it contains only a single addition of the relocation constant r. In general, if the expression can be rearranged to result in the addition of either of the following, it is legal:

0*r	absolute expression
1*r	relocatable expression

If the rearrangement results in the following, it is illegal:

n*r	where n is not 0 or 1

Also, if the expression involves r to any power other than 1, it is illegal. This leads to the following rules:

1. Only two values of relocatability for a complete expression are allowed (ie. n*r where n = 0 or 1).
2. Division by a relocatable value is illegal.
3. Two relocatable values may not be multiplied together.
4. Relocatable values may not be combined by logical operators.
5. A relocatable value may not be logically shifted.

If any of these rules is broken, the expression is illegal and an error message is given.

If X, Y, and Z are relocatable symbols, then:

X+Y-Z	is relocatable
X-Z	is absolute
X+7	is relocatable
3*X-Y-Z	is relocatable

4&X-Z   is illegal

Only 16-bit quantities may be relocated. All 8-bit values must be absolute or an error will be given.

Relocation Bases

One of the unique capabilities of the PSA Macro Assembler is its ability to handle symbolic references to separately located areas of memory, where the mapping of symbols to physical addresses occurs at linkage edit time. The symbolic names for independently located memory areas are called "relocation bases". These relocation bases may represent ROM vs. RAM, shared COMMON areas, special memory areas such as video refresh, memory mapped I/O, etc. Within each subprogram, each of these memory areas is referenced by a unique name, with the actual allocation deferred to the link edit and load process. All memory references within the assembled program are relative to one of these relocation bases.

As each relocation base is assigned a name in the program (through the use of the .EXTERN pseudo-op), it is implicitely assigned a sequential identifying number. This number appears in the listing as part of any address relative to that base.

Four of these relocation bases (0-3) have predefined names and meanings, and are treated differently at linkage edit time than the remainder of the bases. Base 0 represents absolute memory locations (ie. it always has the value of 0). Base 1 has the name .PROG. and represents the program area (maybe PROM or ROM). Most program code (and data in non-rommed programs) is generated relative to this relocation base. Base 2 has the name .DATA. and represents the local data area for each module. Most local data is defined relative to this base. Base 3 has the name .BLNK. and represents the global "blank common". This relocation base is always assigned the value of the first free byte in memory after the local data storage (.DATA.) and other data relocation segments by the linkage editor. Because it is always the last allocated, modules referencing this area can be included in any order, regardless of the amount of the area they use.

Relocation segments relative to bases 1 and 2 (.PROG. and .DATA.) are always allocated additively (ie. after each module is allocated, the value of the relocation base is increased by the size of the segment). All other relocation bases are normally assumed to have constant values during the allocation process (usually assigned by the linkage editor).

Each symbol defined during the assembly has a relocation base associated with it. There are no limitations on inter-base references (ie. code relative to .PROG. can freely reference data relative to .DATA.). Expressions containing symbols must evaluate to a value relative to a single relocation base, but may contain references to multiple relocation bases. All relocation base references except for the final result must be part of sub-expressions which evaluate to absolute values. For example, if T and U are symbols relative to base 1, V and W relative to base 2, and X and Y relative to base 3, then the following are valid expressions:

```
T+(V-W)          (note the parentheses to make V-W
                  a subexpression)
X+3
T-(V-W)*U+(X-Y)
```

and the following are invalid:

```
T+U              (within a relocation base, the
                  normal relocation rules apply)
T+V-W            (T+V is the first subexpression,
                  and it is mixed relocation bases)
```

It should be noted that conceptually, normal external symbols are simply relocation bases with a size of zero (0), and the assembler treats them that way. An assignment of the form:

```
N==P+5
```

where P is an external symbol, makes N a symbol whose address is relative to P, even though P has no size. Hence, expressions of the form:

```
5*(P-N)
```

where P and N have the same relocation base, are in fact valid.

Chapter 3

Pseudo-Operations

Pseudo-operations (pseudo-ops) are directions to the assembler to perform certain operations for the programmer, as opposed to machine operations which are instructions to the computer. Pseudo-ops perform such functions as listing control, data conversion, or storage allocation.

Address Mode and Origin

The PSA Macro Assembler normally assembles programs in relocatable mode, so that the resultant program can be loaded anywhere in memory for execution. Therefore, all programs are assembled assuming their first byte is at address zero (0), because they can be relocated anywhere. When desired, however, the assembler will generate absolute object code, either for the entire program, or just selected portions. The assembler will also locate the assembled code at any address desired. The two pseudo-ops which control address mode, relocation base and address origin are .LOC and .RELOC.

.LOC n

This statement sets the location counter to the value n, which may be any valid expression. If n is an absolute value, then the assembler will assign absolute addresses to all of the instructions and data which follow. If n is relocatable, then relocatable addresses will be assigned, relative to the relocation base of the expression.

The program is assumed to to start with an implicit .LOC to relocatable address zero (0) of the relocation base named .PROG. (the default relocation base for normal programs). A program can contain more than one .LOC, each controls the assignment of addresses to the statements following it.

To reset the program counter to its value prior to the last .LOC, the statement:

.RELOC

is used. This statement restores both the value, the relocation base and the addressing mode which were in effect before the immediately preceding .LOC. If no .LOC has been done, then a .RELOC is equivalent to a:

.LOC 0

When in relocatable addressing mode, the assembler will determine whether each 16-bit value is absolute or relocatable as described in Chapter 2.

Data Definition

The PSA Macro Assembler provides a number of different pseudo-ops for describing and entering data to be used by the program.

.RADIX

When the assembler encounters a number in a statement, it converts it to a 16-bit binary value according to the radix indicated by the programmer. The statement:

```
.RADIX n
```

where n is 2, 8, 10, or 16, sets the radix to n for all numbers which follow, unless another .RADIX statement is encountered, or the radix is modified by the ^r operator or a suffix radix modifier.

The statement:

```
.RADIX 10
```

implicitly begins each assembly program, setting the intial radix to decimal.

.BYTE

To enter one (or more) 8-bit (one byte) data values into the program, the statement:

```
.BYTE {[r]}n {, {[r]}n ...}
```

where r is an optional repeat count which can be any 16-bit value (including 0) and n is any expression with a valid 8-bit value, is used. More than one byte can be defined at a time by separating it from the preceding value with a comma. All of the bytes defined in a single .BYTE statement are assigned consecutive memory locations. For example:

```
.BYTE 23,4*^HOFF,BETA-ALPHA,[5]0
```

defines three sequential bytes of data.

.WORD

To enter a 16-bit (two byte) value into the program, the statement:

```
.WORD nn {, nn ...}
```

where nn is any expression with a valid 16-bit value, is used. Multiple 16-bit values may be defined with one .WORD statement by separating each from the preceding one with a comma.

All 16-bit values defined by the .WORD pseudo-op are stored in standard Z80 word format, least significant byte first.

For example, the following statement:

```
.WORD ALPHA,234*BETA,^H0EEFF
```

defines three sequential 16-bit values, or a total of six bytes of data.

.ASCII, .ASCIZ, and .ASCIS

To enter strings of text characters into the program, one of the statements:

```
.ASCII s | [n]
.ASCIZ s | [n]
.ASCIS s | [n]
```

is used. The s represents any valid string value. Each character in the text is converted to its 7-bit ASCII representation (with the eighth bit zero), and stored in sequential memory locations. The string value may be followed by another string value (seperated by an optional comma), and this may be repeated as desired.

If it is necessary to include values in the text string for which no character exists, then the second option shown above may be used. If in place of a string value, the assembler finds a left square bracket ([), then the numeric expression enclosed within it and a matching right square bracket (]) is evaluated as an 8-bit value and stored as the next byte of the string. These 8-bit values may be intermixed with string values as required (optionally seperated by commas). Note that this precludes the use of bracketed delimited strings as string values in these statements.

It is important to note that tab, carriage return, and line feed are all valid characters within a string value. It is therefore possible that a .ASCIx statement will encompass more than one line in the source program.

The difference between the three pseudo-ops described above is in their treatment of the last byte generated by the statement. The .ASCII statement just stores the byte. The .ASCIZ statement stores one additional byte after the last one, a null (zero) byte to mark the end of the string in memory. The .ASCIS pseudo-op sets the high-order (eighth) bit of the last byte to one to flag the last byte.

The following are all valid .ASCIx statements:

```
.ASCII /This is a string/
.ASCIZ /This is two/ ' strings in one place'
.ASCIS [^H0D] [^H0A] "Message on new line"
.ASCII \
Message on new line\
.ASCII PNAME .string variable plus string.
.ASCIZ <[2,B]"substring expression"
```

.RAD40

The Radix-40 character set for symbols was chosen because it allows a six character symbol to be stored in only four bytes of memory. To allow the program to define data bytes in this character set, the statement:

```
.RAD40 symbol1 {, symbol2 ...}
```

is used. The symbol must conform to all the rules specified for assembler symbols, and is converted into the Radix-40 notation and stored in four sequential bytes of memory. If multiple symbols are to be converted and stored, each must be separated from the preceding one by a comma.

Storage Allocation

The PSA Macro Assembler allows the programmer to reserve single locations, or blocks of many locations, for use during the execution of the program. The two pseudo-ops used for this purpose are .BLKB and .BLKW. The format of the statement using these pseudo-ops is:

```
.BLKx n
```

where n is the number of storage locations to be reserved.

For the .BLKB pseudo-op, each storage location consists of one byte, so the above statement will reserve n contiguous bytes of memory, starting at the current location counter. The .BLKW pseudo-op uses a word (two bytes) as its storage unit, so the above statement would reserve n words, or two times n bytes of contiguous memory.

For example, each of the following statements reserves 24 (decimal) bytes of storage:

```
.BLKB 24.
.BLKW ^D12
.BLKB 2*12.
```

Temporary Variables

The PSA Macro Assembler allows the allocation of temporary variable arrays for use during the assembly process. These arrays may be used both on a global basis, and within macro expansions. The use of these arrays in each of these contexts is described elsewhere in this manual. The actual allocation of space for these arrays is done using the .TEMPS pseudo-op. The format of this statement is:

```
.TEMPS n
```

where n is a 16-bit value specifying the number of array elements to allocate.

This statement may appear anywhere in the program, and may be used more than once. In either context (global or macro), the reappearance of the statement results in the reallocation of the array space. A value of zero results in the freeing of all allocated space. This space allocation is dynamic, and the freed space may be used by other parts of the assembly process.

Program Termination

Every program must be terminated by a .END pseudo-op. The format of this statement is:

```
.END start
```

where start is optional an starting address for the program. The starting address is normally only necessary for the main program. Subprograms, which are called from the main program, need no starting address.

When the assembler encounters the .END pseudo-op during pass 1 of the assembly, it initiates pass 2 of the assembly. On a listing pass, the .END pseudo-op initiates the printing of the symbol table (if not suppressed by a prior .XSYM pseudo-op). On a object producing pass, the .END pseudo-op output the EOF record to the object file.

Subprogram Linkage

Programs usually consist of a main program and numerous subroutines which communicate with each other through parameter linkages and through reference to symbols defined elsewhere in the program. Since the PSA Macro Assembler provides the means for the various program components to be assembled separately from each other, the linkage editor (which finally puts the pieces together) must be able to identify those symbols which are references (or referenced) external to the current program. For a given subprogram, these "linkage" symbols are either symbols defined internally which must be available to other programs to reference, or symbols used internally but defined externally to the program. Symbols defined within the program but available to other subprograms are called "internal" symbols. Symbols used internally but defined elsewhere are called "external" symbols.

To set up these linkages between subprograms, four pseudo-ops are provided: .IDENT, .EXTERN, .INTERN, and .ENTRY.

The .IDENT statement has the format:

```
.IDENT symbol
```

where symbol is the relocatable module name. This name is used by the linkage editor to identify the module on memory allocation maps, and to allow the selective loading of the module if it is part of a subprogram library. If the .IDENT statement does not appear in a program, the name ".MAIN." is assumed. The .IDENT name appears at the top of every listing page, and is displayed on the console at the start of the second assembly pass of that module.

All three remaining statements have the same format:

```
.EXTERN symbol1 {, symbol2 ...}
.INTERN symbol1 {, symbol2 ...}
.ENTRY symbol1 {, symbol2 ...}
```

where symbol1 is the symbol being declared as external, internal, or as an entry point. Multiple symbols may be declared in the same statement by separating each from the preceding one with a comma.

The .EXTERN statement identifies symbols which are defined elsewhere. External symbols must not be defined within the current subprogram. External symbols may be used in the same manner as any other relocatable symbol, with the following restrictions:

1. The use of more than one external symbol in a single expression is illegal. Thus V+W where V and W are both external is illegal.
2. Externals may only be additive. Therefore the following expressions are illegal (where V is an external symbol):

   -V
   2*V
   SQR-V
   2*V-V

3. The final evaluation of the expression containing the external reference must result in external symbol + (or -) constant value. Furthermore, if this expression is used as an 8-bit (byte) value, the constant must be between -128 and +255 or an error will result.

Symbols declared as external by the .EXTERN pseudo-op may also be used as relocation bases. This is done by using an external symbol as the argument to a .LOC pseudo-op. All memory allocated by the assembler after the .LOC will be addressed relative to the specified relocation base. The most common use of this capability is the declaration of COMMON blocks for the sharing of data between assembler and FORTRAN subprograms. All named COMMON blocks are in fact just different relocation bases. Symbols used as relocation bases have unique values during the assembly of the program module. At any point in time, the current value of the relocation base symbol is the number of bytes which have been allocated to that base so far. This means that subsequent .LOC pseudo-ops referencing the same external symbol will start the memory allocation at the next available byte in that relocation base, not at relative location zero (0).

There are three predefined relocation base symbols: .PROG., .DATA. and .BLNK.. These relocation bases are used for the program code, seperately located data (in a ROM/RAM environment), and blank (unnamed) common respectively.

The .INTERN pseudo-op identifies those symbols within the current subprogram which are to be made accessible to other programs as external symbols. This statement has no effect on the assembly process for the current program, but merely records the name and value of the identified symbols on the object tape for later use by the linkage editor. An internal symbol must be defined within the current program as a label, or in a direct assignment statement.

The .ENTRY pseudo-op functions identically to the .INTERN pseudo-op, with one addition. It is sometimes desirable to put many subroutines with common usage into one "library", and to allow the linkage editor to select only those programs from the library which are called by the program being linkage edited.

The .ENTRY statement, in addition to functioning as a .INTERN statement, also flags the specified symbols as program entry points. If the subprogram is later put into a library, this will specify to the linkage editor that this program is to be included only if one of its entry points is referenced as an external symbol by an already included program.

Since these entry points are external to the program referencing them, they must be listed in a .EXTERN statement in the calling program.

Program Identification

To facilitate the identification of various program modules manipulated by the programmer, the PSA Macro Assembler provides a unique program identification capability. The format of the pseudo-op controlling this feature is:

```
.PROGID id,ver,rev
```

where id is any valid assmbler symbol uniquely identifying the program module, ver is an 8-bit value specifying the program version, and rev is an 8-bit value specifying the program revision. This information is ouput to the object file in a special identification record which has no effect on the linkage edit process (as opposed to the .IDENT pseudo-op). The information provided by this pseudo-op is listed on the map listing presented by the linkage editor so as to identify which particular module was actually used in the linkage process.

Listing Control

Program listings are printed on the list device during the second asembly pass if enabled. The listing is printed as the source program statements are processed during the pass. The standard listing contains (from left to right):

1. Error flags (if present).
2. Location counter for the first byte generated by this statement.
3. Instruction or data in hexadecimal (maximum of six bytes per line printed).
4. Exact image of the input statement.

The standard listing displays all 16-bit quantities in 16-bit (two byte), most significant byte first, format. These quantities are properly reversed in the object code as required by the Z80. A 16-bit relocatable address relative to the .PROG. relocation base is flagged with an apostrophe (′), one relative to the .DATA. relocation base is flagged with an asterisk (*), and all others are followed by the assigned number of their relocation base.

All 8-bit (byte) quantities are displayed as 8-bit (byte) values except when relative to an external symbol (or relocation base), in which case they are followed by the assigned number of the external or relocation base.

Within a macro expansion, only the macro call and those statements which generate actual object code are normally listed.

If a single statement generates more than the maximum of five bytes that can be listed on a single line, the remaining bytes are properly generated, but not normally listed.

A listing always begins at the top line of the page, and 60 lines are printed per page, with a two line margin at the top, and a two line margin at the bottom. Each page is numbered, and can have an optional title and sub-title.

The standard listing options can be changed and expanded by the use of the following pseudo-operations:

.PAGE w {, l} This statement controls the page width and length used for the assembly listing. The width is w, any 8-bit value, and the optional length is l, also an 8-bit value. This change takes effect immediately.

.PAGE This statement causes the assembler to skip to the top of the next page (by counting lines). A form feed character in the input text will have the same effect.

.XLIST This statement causes the assembler to stop listing the assembled program at this point.

.LIST This statement is normally used following a .XLIST to resume program listing.

.LALL This statement causes the assembler to list everything which is processed. This includes all text, macro expansions, and all other statements normally suppressed in the standard listing.

.XALL This statement is normally used following a .LALL to resume the normal listing.

.SALL This statement causes the suppression of all macro expansions and their text. It can be reset by a subsequent .LALL or .XALL.

.XSYM This statement suppresses the symbol table listing normally performed upon encountering the .END statement.

.LSYM Normally not used, this statement enables the listing of the symbol table previously suppressed by the .XSYM pseudo-op.

.LADDR This statement causes the assembler to list all 16-bit quantities in the same order it generates them in the object code (least significant byte first).

.XADDR Normally used following a .LADDR statement, this statement resumes the normal listing of 16-bit quantities in non-swapped format.

.LIMAGE This statement causes the assembler to list every byte generated, even if more than one line (at six bytes per line) is required. In this mode, the assembler will attempt to split the input source statement to indicate which part of the statement is generating which bytes.

.XIMAGE Normally used following a .LIMAGE statement, this statement resumes the normal listing of only six bytes of generated data per statement.

.LCTL This statement causes all subsequent listing control statements (eg. .XLIST) to be listed themselves. Normally, no listing control statement is itself listed. The .XCTL pseudo-op is used to reset this option.

.XCTL Normally used following a .LCTL statement, this statement resumes the default suppression of the listing of listing control statements.

.SLIST This statement causes the current listing control flags to be saved on a four element push-down stack. The current flags settings remain unchanged. These setting may later be restored with the .RLIST pseudo-op. This pseudo-op may be followed on the same line with another listing control pseudo-op, which will take effect prior to the listing of the .SLIST statement.

.RLIST This statement restores the listing control flags from the top element of the .SLIST push-down stack. These new flags take effect with the statement following the .RLIST.

.TITLE svalue This statement defines the string value to be the title to be printed at the top of every page of the listing. The actual text of the title must be no longer than 72 characters. If the .TITLE pseudo op is the first statement on a page, then the new title will be printed at the top of that page.

.SBTTL svalue This statement defines the string value to be the sub-title to be printed at the top of every page of the listing. It follows the same rules as the .TITLE pseudo-op.

.REMARK svalue This statement inserts a remark into the listing. The string value can be any number of lines long.

.PRNTX svalue This statement, when encountered, causes the string value to be typed on the console. This statement is frequently used to print out conditional information, and to report the progress through pass 1 on very long assemblies.

Object Control

The PSA Macro Assembler normally produces an object file in the TDL Standard Relocatable Format (see Appendix D). However, the assembler can produce an object file compatible with the INTEL Standard Absolute Hex Format. To control which format is being produced, the two pseudo-ops .PREL and .PABS are used. The .PABS pseudo-op causes the assembler to produce an INTEL compatible file for all following generated code. The .PREL causes the assembler to return to producing a TDL Format file.

Every program starts with an implicit .PREL pseudo-op.

In addition, the assembler can output the object file in both binary and ASCII. To control which type of output is being produced, the two pseudo-ops .PBIN and .PHEX are used. The .PBIN pseudo-op causes the assembler to produce a binary output in the current format. The .PHEX pseudo-op causes the output of an ASCII file. Every program starts with an implicit .PHEX pseudo-op.

To control the generation of linkable object modules, two pseudo-ops are provided. The .LINK pseudo-op indicates that linkage information is to be included in the object file produced. The .XLINK pseudo-op inhibits this information from being output. Every program starts with an implicit .XLINK pseudo-op.

The PSA Macro Assembler provides one additional facility to assist the PSA Z80 Debugging System (BUG). At the programmers option, the assembler will output all of the global (non-local) symbols in the program module into the end of the object file. For each symbol, the assembler also punches its relocation base and its value relative to that base. Two pseudo-ops are provided to control this symbol table output. The .PSYM pseudo-op enables the output, and the .XPSYM pseudo-op disables it. The default is to not output the symbol table (.XPSYM).

Conditional Assembly

Parts of a program may be assembled on a conditonal basis depending on the results of certain tests specified to the assembler through the use of the .IFx pseudo-op.

The general form of the pseudo-op is:

```
.IFx arg,[true text] ... {[false text]}
```

where the text within the first square brackets is assembled only if the specified test on the argument is TRUE, and the optional text within the second set of brackets is assembled if the condition is false. Any number of spaces or blank lines (or lines with only comments) may seperate the true and false texts.

The square brackets around the true text may be omitted if there is no false text, and the entire true text is contained on the same line as the .IFx pseudo-op.

The first set of conditions which can be tested are the numeric value of the argument. These pseudo-ops are listed below:

```
.IFE  n,[...]        TRUE if n=0 or n=blank
.IFN  n,[...]        TRUE if n<0 or n>0
```

```
.IFG  n,[...]      TRUE if n>0
.IFGE n,[...]      TRUE if n>0 or n=0
.IFL  n,[...]      TRUE if n<0
.IFLE n,[...]      TRUE if n<0 or n=0
```

The following .IF pseudo-ops test for whether the assembler is processing pass 1 or not:

```
.IF1 ,[...]        TRUE if it is pass 1
.IF2 ,[...]        TRUE if it is not pass 1
```

The next set of conditionals tests for whether a symbol has been defined yet or not:

```
.IFDEF  symbol,[...]    TRUE if the symbol is defined
.IFNDEF symbol,[...]    TRUE if the symbol is undefined
```

The next set of .IF pseudo-ops tests to see whether its string value argument is blank or not. The format is as follows:

```
.IFB  s,[...] TRUE if s is blank
.IFNB s,[...] TRUE if s is not blank
```

The string value is blank if it is empty, or consists only of spaces and tabs.

The last pair of conditionals operate on string values. They take two string value arguments and make a character by character comparison of the two strings to determine if the condition is met. The format of these conditionals is as follows:

```
.IFIDN s1 s2,[...] TRUE if s1 identical to s2
.IFDIF s1 s2,[...] TRUE if s1 different from s2
```

The maximum length of the strings to be compared is 255 characters. In making the comparison, all trailing blanks and tabs are ignored in the two arguments.

Synonyms

It sometimes becomes useful, for documentation or ease of programming, to define new names for already existing symbols. The PSA Macro Assembler has four pseudo-ops which allow the definition of synonyms for already defined symbols. The format of these pseudo-ops is:

```
.xxSYN symbol1,symbol2
```

The four pseudo-ops are .SYN, .OPSYN, .SYSYN, and .MASYN. The only difference between the four is that the latter three limit the type of symbol for which the synonym is being defined.

The statement above defines the second operand as being synonymous with the first operand. In the case of the .SYN pseudo-op, the symbol tables are searched for the first operand in the order: programmer defined symbol, macro, operation. The .OPSYN pseudo-op limits the search to operations, the .SYSYN to programmer defined symbols, and the .MASYN to macros. The second operand is defined to be identical to the first operand at the time the synonym is defined. Later changes to the first operand will not affect the second.

The following are valid synonym definitions:

```
.OPSYN .BYTE,DB
.SYN .WORD,DW
.SYSYN ALPHA,BETA
.SYN A,R1
```

Object Machine Validation

Although the PSA Macro Assembler will run only on a Z80 compatible processor, it can obviously be used to generate object code for any of the 8080 compatible micro-processors. To facilitate the use of the assembler for this purpose, two additional pseudo-ops are available: .I8080 and .Z80.

The .I8080 pseudo-op causes all subsequent uses of machine operations which are unique to the Z80 (and hence unavailable on the 8080) to be flagged with a Z warning message. Such uses will be properly assembled however.

The .Z80 pseudo-op (which is the default) disables the feature so that no further Z warnings will be given.

Library File Generation

It is often desirable to maintain a related set of independent object modules as a single source and object file for later use with the library search facility of the PSA Linkage Editor (LINK). To facilitate this the .PRGEND pseudo-op can be used. The format is:

```
.PRGEND
```

This pseudo-op functions identically to the .END pseudo-op, except that, after completing the assembly of the current module, the assembler continues with another module following. Multiple modules assembled in this manner from a single source file produce a single object file which can be linked in library search mode, and a single listing. Each module assembly is completely independent however. The last module in the source file must be terminated by a .END pseudo-op, not a .PRGEND.

Library Source File Usage

It is often convienent to be able to utilize the same section of assembler source code in a number of different assemblies. The .INSERT pseudo-op allows this to be done easily. The format is:

```
.INSERT {d:}file{.ext}
```

where d is the optional PDOS disk specifier (defaulting to the source file disk), file is the desired file name, and ext is the optional file extension (defaulting to ASM).

This pseudo-op causes the specified file to be copied into the assembly in its entirety, and to be treated exactly as if it were part of the original source file. All inserted source is flagged with an "@" on the listing. Only one level of .INSERT is allowed, they cannot be nested.

This pseudo-op will generate an "F" error if the file is not found, incorrectly specified, or if an .INSERT is already in progress.

Date and Time Reference

It is often desirable to include the date and time of assembly in the text of various messages within a program. The PSA Macro Assembler allows this through the addition of two new pseudo-ops. These are: .DATE and .TIME. These pseudo-ops take no arguments. They simply cause the generation of the specified ASCII string at that location in the program. For date, it is MM/DD/YY, and for time, it is HH:MM:SS. Each pseudo-op generates 8 bytes of information. If these pseudo-ops are used on a PDOS compatible operating system which does not support time and/or date, then 8 bytes of spaces will be generated by the pseudo-op.

Chapter 4

Macros

A common characteristic of assembly language programs is that many coding sequences are repeated over and over with only a change in one or two of the operands. It is convenient, therefore, to provide a mechanism by which the repeated sequences can be generated by a single statement. The PSA Macro Assembler provides the capability to do so by allowing the repeated sequences to be written, with dummy values for the changed operands, as a macro. A single statement, referring to the macro by name and providing values for the dummy operands, can then generate the repeated sequence.

Macro Definition

A macro is defined by use of the .DEFINE pseudo-op. This is followed by the symbolic name of the macro. The macro name must follow the rules for the construction of symbols. The name may be followed by a list of dummy arguments enclosed in square brackets. The dummy arguments are separated by commas, and may be any symbol which is convenient. Following the macro name and optional dummy arguments must be an equal sign (=). The following are examples of the heading part of a macro definition:

```
.DEFINE MACRO =
.DEFINE MOVE[A,B] =
.DEFINE BIGMAC[ARG1,ARG2,ARG3,%ARG5] =
```

Following the macro definition header comes the body of the macro. It need not start on the same line as the definition header. The body of the macro is delimited by a matched pair of left and right square brackets ([]). For example:

```
.DEFINE MOVE[A,B]=
[LDA A
STA B]
```

Macro Calls

A macro may be called by any statement. A macro call consists of the macro name followed (optionally) by a list of arguments. The arguments are separated by commas, and may optionally be enclosed in left and right square brackets ([]). If the brackets are used (the first non-blank/non-tab character after the macro name is a left square bracket), then the arguments are terminated by a right square bracket. If there are n dummy arguments in the macro definition, then all arguments after the first n are ignored (although they do take space and time to process). If the brackets are omitted, the argument string ends when a carriage return or semicolon is encountered.

The arguments must be written in the order in which they are to be substituted for the dummy arguments. The first argument is substituted for each appearance of the first dummy argument, the second for the second, etc. The actual arguments are substituted as character strings for the dummy arguments, no evaluation of the arguments takes place until the macro is processed.

Referring to the definition of MOVE above, the occurrence of the statement:

```
MOVE ALPHA,BETA
```

will cause the substitution of ALPHA for A and BETA for B in the macro.

Statements which contain macro calls may be labelled and have comments like any other statement.

Macro arguments are terminated only by comma, carriage return, semicolon, or right square bracket (when started by left square bracket). These characters may not be used in the arguments unless the argument is enclosed in parentheses. Each time an argument is passed to a macro, one set of matched parentheses is removed, but all of the characters within the parentheses are substituted for the dummy argument in the macro. Trailing spaces and tabs at the end of a macro argument (not enclosed in parentheses) are ignored.

A substring expression may be used as an argument to a macro, and if not enclosed in paretheses, will be evaluated before the macro is called.

Macros do not need to have arguments. The macro name (and arguments if any) may appear anywhere in a statement where a symbol would normally appear, and the text of the macro exactly replaces the macro name and its arguments in that statement.

Comments

Comments may be included within a macro definition. Storing the comments with the macro (so that they will appear when the macro is expanded) takes space however. If the comment within the macro definition is preceded by two semicolons (instead of the normal one), the comment will be ignored during the definition of the macro, and will not be stored as part of the definition. This will eliminate the appearance of the comment every time the macro expansion is listed, however.

Created Symbols

When a macro is called, it is often useful to generate symbols without explicitly stating them in the call. A good example of this is labels within the macro body. It is usually not necessary to refer to these label externally to the macro expansion, therefore there is no reason why the programmer should be concerned as to what those labels are. The same with temporary data areas. To avoid conflicts, however, it is necessary that a different symbol be used each time the macro is called (even with local symbols, the macro could be called more than once between two global symbols). Created symbols are used for this purpose.

Each time a macro that requires a created symbol is called, a symbol is generated and inserted into the macro. These symbols are of the form ..nnnn (two periods followed by four digits). It should be noted that this makes these symbols local symbols (start with two periods). The programmer is advised not to use symbols of this form. The four digits start at 0000 and are incremented by one each time a symbol is created.

A created symbol is specified in the macro definition by preceding a dummy argument by a percent sign (%). When the macro is called, all dummy arguments of the form %symbol are replaced by created symbols (each with a different one). If, however, the position of the dummy argument in the argument list corresponds to an actual argument provided in the call, then the actual argument is used in place of the created one.

An actual argument can in fact be empty (signified by two consecutive commas in the argument list). An argument of this kind (a "null" argument) is considered to be defined as having a value of the empty string (no characters), and will prevent the generation of a created symbol for its corresponding dummy argument.

For example:

```
.DEFINE PRINT[A,%B]=
[CALL LINPRT
JMP %B
.ASCIS \A\
%B:]
```

This macro prints a message on the printer. The first argument to the macro is the text string to be printed. LINPRT is a line printer routine. Labelling the location following the text is necessary because of the indeterminate length of the message. The use of a created symbol here is useful since there would normally be no reason to reference the label. Calling the macro by:

```
PRINT This is the message
```

would result in printing "This is the message" when the assembled macro was executed. If it had been called:

```
PRINT This is the message,MAIN
```

the message would have been printed, but control would be transfered to the label MAIN, which substituted for %B instead of a created symbol.

Concatenation

The apostrophe or single quote (′) is defined within a macro definition as the concatenation operator. This allows a macro argument to be only part of a symbol or expression, with the character string which is substituted for the dummy argument being joined with other character strings that are part of the macro definition to form a complete symbol or expression. This joining is called concatenation. Concatenation is performed by the assembler when an apostrophe is used between the strings to be joined (one or both of which must be a dummy macro argument). For example:

```
.DEFINE BR[A,B]=
[JR'A B]
```

defines a conditional branch statement. When called, the argument A is appended to the JR to form a single symbol. If the call were:

```
BR Z,LOOP
```

then the generated code would be:

```
JRZ LOOP
```

Default Arguments

Normally, missing arguments in a macro are replaced by nulls. For example, in the macro:

```
.DEFINE BYTES[A1,A2,A3,A4,A5,A6]=
[.BYTE A1,A2,A3,A4,A5,A6]
```

a call of BYTES[1,2] would generate an error because of the missing arguments to the pseudo-op .BYTE.

To remedy this, the assembler provides the programmer with the means to supply default arguments to be used when no argument is provided in the macro call. Default argument are defined as part of the macro definition by enclosing them in parentheses and inserting them immediately after the dummy argument to which they refer. To solve the above problem, the definition would be written as:

```
.DEFINE BYTES[A1(0),A2(0),A3(0),A4(0),A5(0),A6(0)]=
[.BYTE A1,A2,A3,A4,A5,A5]
```

which would always generate six bytes of data, regardless of how many arguments were provided in the call.

ASCII Interpretation of Numeric Arguments

If the reverse slash (\) preceeds the first character of an argument in a macro call, the value of the expression following the reverse slash is converted to an ASCII string. This string is then used as the argument to the call. The value is considered to be a 16-bit positive value, and the conversion is done in the current radix. Leading zeros are suppressed unless the value is zero.

For example:

```
V = 5
W = 6
MACRO \V+W, \V*W
```

is the same as:

```
MACRO 11, 30
```

if the current radix is 10.

Macro Expansion Termination

Under normal conditions, a macro expansion terminates at the end of the macro definition. It is sometimes desirable to terminate the macro expansion prior to the end of the definition. This is usually done as part of some conditional assembly within the macro. A special pseudo-op is provided for this purpose:

```
.EXIT
```

When processed by the assembler, the .EXIT pseudo-op immediately terminates the macro expansion, just as if the end of the macro had been encountered. Only the current expansion is terminated if multiple macro expansions are being nested.

User Defined Macro Errors

It is sometimes desirable to have a macro cause an assembly error. This might be done when invalid parameters are passed to the macro, or if parameters are missing. A special pseudo-op is provided to allow this:

```
.ERROR string value
```

This pseudo-op will cause an asterisk (*) to be listed as the error code, the error count to be incremented by one, and the line to be listed as an error. The string value is treated exactly as in a .REMARK pseudo-op, and can be used to provide information about the nature of the error.

Nesting

Macros may be nested. This means that macros may be both called and defined within other macros. A macro that is defined within another macro may not be called until the defining macro has been called. At that time, the new macro is available to be called by any statement.

The only limit to how many levels deep macro calls and definitions may be nested is the amount of memory available.

Macro Labels and Branching

It is quite often desirable to process different parts of the macro definition depending on certain conditions, or to repeatitively process one part over again. To provide for both abilities, macro labels and branches are provided.

A macro label is specified in the same manner as a normal label, except that it is followed by a right angle bracket (>) instead of a colon (:). There may be multiple macro labels per line, and a macro label may exist on a line by itself. Macro labels have no effect on the assembly of the macro unless explicitely referenced by a macro branch pseudo-op.

A macro branch pseudo-op is specified as follows

```
.GOTO mlabel
```

where mlabel is a macro label defined within the current macro definition. When encountered during the assembly process, the specified macro label is searched for, and if found, the assembly process continues from that point in the macro text. The search begins at the start of the macro text, so both forward and backward branches are allowed. If the macro label is not found, an error is given, and the macro expansion is terminated. To specifiy conditional branching, the macro branch pseudo-op is used within a single-line conditional statement.

For example:

```
I=0
MLABEL>
  .WORD ARG1
  .BYTE ARG2
I=I+1
.IFL I-ARG3,.GOTO MLABEL
```

would assemble the specified sequence ARG3 times.

Local Temporary Variables

It is often desirable to have available local symbols to store numeric values during the expansion of a macro. The PSA Macro Assembler provides local temporary variables for this purpose. Local temporary variables function similarly to the global variable array described earlier. The number of local array elements is specified through the .TEMPS pseudo-op. These local elements are specific to the current macro expansion only. If the macro is expanded recursively, new locations will be used, and all old values will be preserved. As long as a given expansion is active, its locals are preserved. Once an expansion is terminated, all space utilized by its local variables is available to other macros. References to local variables are made using the format "![sub]". The rules governing these locals are the same as for the globals. They may be used as symbols and as labels. It is important to note, however, that all local variables are initialized to a value of absolute zero, and the use of a local as a label is only valid if the reference to the label occurs after the definition.

For example:

```
  .TEMPS 2
![0]=0
MLABEL>
![1]:
  SUI ARG1
  JRNC ![1]
  ADI ARG1
```

```
![0]=![0]+1
.IFL ![0]-ARG2,.GOTO MLABEL
```

Special Variables

To ease the processing of macros with variable numbers of arguments, a special symbol & (ampersand) has been defined. This symbol may be used in any arithmetic expression where a normal symbol is allowed, and always has the value of the number of arguments passed to the current macro invocation.

Subscripted Argument Reference

It is not always possible to predetermine the number of arguments that will be passed to a macro prior to its invocation. It is cumbersome to have to reference each passed argument by name, requiring a seperate section of the macro definition to handle each one. For this reason, macro arguments may alternately referenced positionallly by use of a subscripted argument reference. The format of this reference is "_marg[sub]". The underscore (_) flags the subscript reference, and marg is the name of a macro argument to be used as the "base" for the subscripting. This allows a macro to have a number of fixed arguments, and then a following number of variable ones. The subscript (sub) is a 8-bit value which indicates that the sub'th argument after marg is being referenced. Note that a zero subscript refers to the named argument. This reference is only triggered by the appearance of an underscore followed by a macro argument, so underscores may appear anywhere in a macro expansion normally. If the subscript references an argument which was not given in the call, a null value is returned. It is not necessary to explicitely name any argument which is going to be referenced by subscript. Only the base argument must be named.

For example:

```
.DEFINE SYMBOL[ARG]=[
  .TEMPS 1
![0]=0
MLABEL>
  .ASCIZ "_ARG[![0]]"
  .WORD $'_ARG[![0]]
![0]=![0]+1
.IFL ![0]-&,.GOTO MLABEL
]
```

generates a symbol table consisting of ASCII strings and branch addresses. The macro might be called:

```
SYMBOL HELP,TEST,EXIT,DISPLAY
```

and would generate:

```
  .ASCIZ "HELP"
  .WORD $HELP
  .ASCIZ "TEST"
```

```
.WORD $TEST
.ASCIZ "EXIT"
.WORD $EXIT
.ASCIZ "DISPLAY"
.WORD $DISPLAY
```

Appendix A

Summary of Machine Operation Mnemonics

The following section presents a summary of the Z80 machine operations and their assembler mnemonics. The appendix is arranged by type of instruction for ease of reference. For further information on the machine operations, refer to the Technical Manual for the Z80 compatible microprocessor which is being used.

To make the information presented more readily usable, a shorthand notation is used for describing the assembler format of the instruction and its actual operation. All capital letters and special charcters in the mnemonic description are required. The lower case letters indicate a class of values which can be inserted in the instruction at that point. A single lower case letter indicates an 8-bit quanity or register, while a double lower case letter indicates a 16-bit quantity or register. A symbol enclosed in parentheses in the machine operation section indicates that the value whose address is specified is used. The following is a summary of the notation used; exceptions will be noted where appropriate in the following sections.

r	one of the 8-bit registers A, B, C, D, E, H, L
n	any 8-bit value
ii	an index register reference, either X or Y
d	an 8-bit index displacement where -128 < d < 127
zz	B for the BC register pair, D for the DE pair
nn	any 16-bit value, absolute or relocatable
rr	B for the BC register pair, D for the DE pair, H for the HL pair, SP for the stack pointer
qq	B for the BC register pair, D for the DE pair, H for the HL pair, PSW for the A/Flag pair
s	any of r (defined above), M, or d(ii)
IFF	interupt flip-flop
CY	carry flip-flop
ZF	zero flag
tt	B for the BC register pair, D for the DE pair, SP for the stack pointer, X for index register IX
uu	B for the BC register pair, D for the DE pair, SP for the stack pointer, Y for index register IY
b	a bit position in an 8-bit byte, where the bits are numbered from right to left 0 to 7
PC	program counter
v[n]	bit n of the 8-bit value or register v
v[n-m]	bits n through m of the 8-bit value or register v
vv\H	the most significant byte of the 16-bit value or register vv
vv\L	the least significant byte of the 16-bit value or register vv
Iv	an input operation on port v
Ov	an output operation on port v
w<-v	the value of w is replaced by the value of v
w<->v	the value of w is exchanged with the value of v

8-Bit Load Group

Mnemonic		Operation	# of Bytes
MOV	r,r'	r <- r'	1
MOV	r,M	r <- (HL)	1
MOV	r,d(ii)	r <- (ii+d)	3
MOV	M,r	(HL) <- r	1
MOV	d(ii),r	(ii+d) <- r	3
MVI	r,n	r <- n	2
MVI	M,n	(HL) <- n	2
MVI	d(ii),n	(ii+d) <- n	4
LDA	nn	A <- (nn)	3
STA	nn	(nn) <- A	3
LDAX	zz	A <- (zz)	1
STAX	zz	(zz) <- A	1
LDAI		A <- I	2
LDAR		A <- R	2
STAI		I <- A	2
STAR		R <- A	2

16-Bit Load Group

Mnemonic		Operation	# of Bytes
LXI	rr,nn	rr <- nn	3
LXI	ii,nn	ii <- nn	4
LBCD	nn	B <- (nn+1) C <- (nn)	4
LDED	nn	D <- (nn+1) E <- (nn)	4
LHLD	nn	H <- (nn+1) L <- (nn)	3
LIXD	nn	IX\H <- (nn+1) IX\L <- (nn)	4
LIYD	nn	IY\H <- (nn+1) IY\L <- (nn)	4
LSPD	nn	SP\H <- (nn+1) SP\L <- (nn)	4
SBCD	nn	(nn+1) <- B (nn) <- C	4
SDED	nn	(nn+1) <- D (nn) <- E	4
SHLD	nn	(nn+1) <- H (nn) <- L	3
SIXD	nn	(nn+1) <- IX\H (nn) <- IX\L	4
SIYD	nn	(nn+1) <- IY\H (nn) <- IY\L	4
SSPD	nn	(nn+1) <- SP\H (nn) <- SP\L	4
SPHL		SP <- HL	1
SPIX		SP <- IX	2
SPIY		SP <- IY	2
PUSH	qq	(SP-1) <- qq\H (SP-2) <- qq\L SP <- SP - 2	1
PUSH	ii	(SP-1) <- ii\H (SP-2) <- ii\L SP <- SP - 2	2
POP	qq	qq\H <- (SP+1) qq\L <- (SP) SP <- SP + 2	1
POP	ii	ii\H <- (SP+1) ii\L <- (SP) SP <- SP + 2	2

Exchange and Block Transfer and Search Group

Mnemonic	Operation	# of bytes
XCHG	HL <-> DE	1
EXAF	PSW <-> PSW'	1
EXX	BCDEHL <-> BCDEHL'	1
XTHL	H <-> (SP+1) L <-> (SP)	1
XTIX	IX\H <-> (SP+1) IX\L <-> (SP)	2
XTIY	IY\H <-> (SP+1) IY\L <-> (SP)	2
LDI	(DE) <- (HL) DE <- DE + 1 HL <- HL + 1 BC <- BC - 1	2
LDIR	repeat LDI until BC=0	2
LDD	(DE) <- (HL) DE <- DE - 1 HL <- HL - 1 BC <- BC - 1	2
LDDR	repeat LDD until BC=0	2
CCI	A - (HL) HL <- HL + 1 BC <- BC - 1	2
CCIR	repeat CCI until A=(HL) or BC=0	2
CCD	A - (HL) HL <- HL - 1 BC <- BC - 1	2
CCDR	repeat CCD until A=(HL) or BC=0	2

8-Bit Arithmetic and Logical Group

Mnemonic		Operation	# of Bytes
ADD	r	A <- A + r	1
ADD	M	A <- A + (HL)	1
ADD	d(ii)	A <- A + (ii+d)	3
ADI	n	A <- A + n	2
ADC	s	A <- A + s + CY	
ACI	n		
SUB	s	A <- A - s	
SUI	n		
SBB	s	A <- A - s - CY	
SBI	n		
ANA	s	A <- A & s	
ANI	n		
ORA	s	A <- A ! s	
ORI	n		
XRA	s	A <- A ^ s	
XRI	n		
CMP	s	A - s	
CPI	n		
INR	s	s <- s + 1	
DCR	s	s <- s - 1	

General Purpose Arithmetic and Control Group

Mnemonic	Operation	# of Bytes
DAA	convert A to packed BCD after an add or subtract of packed BCD operands	1
CMA	A <- #A	1
NEG	A <- -A	2
CMC	CY <- #CY	1
STC	CY <- 1	1
NOP	no operation	1
HLT	halt	1
DI	IFF <- 0	1
EI	IFF <- 1	1
IM0	interrupt mode 0	2
IM1	interrupt mode 1	2
IM2	interrupt mode 2	2

16-Bit Arithmetic Group

Mnemonic		Operation	# of Bytes
DAD	rr	HL <- HL + rr	1
DADC	rr	HL <- HL + rr + CY	2
DSBC	rr	HL <- HL - rr - CY	2
DADX	tt	IX <- IX + tt	2
DADY	uu	IY <- IY + uu	2
INX	rr	rr <- rr + 1	1
INX	ii	ii <- ii + 1	2
DCX	rr	rr <- rr - 1	1
DCX	ii	ii <- ii - 1	2

Rotate and Shift Group

Mnemonic	Operation	# of Bytes
RLC	A[n+1] <- A[n] A[0] <- A[7] CY <- A[7]	1
RAL	A[n+1] <- A[n] A[0] <- CY CY <- A[7]	1
RRC	A[n] <- A[n+1] A[7] <- A[0] CY <- A[0]	1
RAR	A[n] <- A[n+1] A[7] <- CY CY <- A[0]	1
RLCR s	s[n+1] <- s[n] s[0] <- s[7] CY <- s[7]	2 (or 4)
RALR s	s[n+1] <- s[n] s[0] <- CY CY <- s[7]	
RRCR s	s[n] <- s[n+1] s[7] <- s[0] CY <- s[0]	
RARR s	s[n] <- s[n+1] s[7] <- CY CY <- s[0]	
SLAR s	s[n+1] <- s[n] s[0] <- 0 CY <- s[7]	
SRAR s	s[n] <- s[n+1] s[7] <- s[7] CY <- s[0]	
SRLR s	s[n] <- s[n+1] s[7] <- 0 CY <- s[0]	
RLD	A[0-3] <- (HL)[4-7] (HL)[4-7] <- (HL)[0-3] (HL)[0-3] <- A[0-3]	2
RRD	(HL)[0-3] <- (HL)[4-7] (HL)[4-7] <- A[0-3] A[0-3] <- (HL)[0-3]	2

Bit Set, Reset, and Test Group

Mnemonic		Operation	# of Bytes
BIT	b,r	ZF <- #r[b]	2
BIT	b,M	ZF <- #(HL)[b]	2
BIT	b,d(ii)	ZF <- #(ii+d)[b]	4
SET	b,s	s[b] <- 1	
RES	b,s	s[b] <- 0	

Jump Group

Mnemonic		Operation	# of Bytes
JMP	nn	PC <- nn	3
JZ	nn	if zero, then JMP else continue	3
JNZ	nn	if not zero	3
JC	nn	if carry	3
JNC	nn	if not carry	3
JPO	nn	if parity odd	3
JPE	nn	if parity even	3
JP	nn	if sign positive	3
JM	nn	if sign negative	3
JO	nn	if overflow	3
JNO	nn	if not overflow	3
JMPR	nn	PC <- nn where -126 < nn-PC < 129	2
JRZ	nn	if zero, then JMPR else continue	2
JRNZ	nn	if not zero	2
JRC	nn	if carry	2
JRNC	nn	if not carry	2
DJNZ	nn	B <- B - 1 if B=0 then continue else JMPR	2
PCHL		PC <- HL	1
PCIX		PC <- IX	2
PCIY		PC <- IY	2

Call and Return Group

Mnemonic		Operation	# of Bytes
CALL	nn	(SP-1) <- PC\H (SP-2) <- PC\L SP <- SP - 2 PC <- nn	3
CZ	nn	if zero, then CALL else continue	3
CNZ	nn	if not zero	3
CC	nn	if carry	3
CNC	nn	if not carry	3
CPO	nn	if parity odd	3
CPE	nn	if parity even	3
CP	nn	if sign positive	3
CM	nn	if sign negative	3
CO	nn	if overflow	3
CNO	nn	if not overflow	3
RET		PC\H <- (SP+1) PC\L <- (SP) SP <- SP + 2	1
RZ		if zero, then RET else continue	1
RNZ		if not zero	1
RC		if carry	1
RNC		if not carry	1
RPO		if parity odd	1
RPE		if parity even	1
RP		if sign positive	1
RM		if sign negative	1
RO		if overflow	1
RNO		if no overflow	1
RETI		return from interrupt	2
RETN		return from non-maskable interrupt	2
RST	n	(SP-1) <- PC\H (SP-2) <- PC\L PC <- 8 * n where 0 <= n < 8	1

Input and Output Group

Mnemonic	Operation	# of Bytes
IN n	A <- In	2
INP r	r <- I(C)	2
INI	(HL) <- I(C) B <- B - 1 HL <- HL + 1	2
INIR	repeat INI until B=0	2
IND	(HL) <- I(C) B <- B - 1 HL <- HL - 1	2
INDR	repeat IND until B=0	2
OUT n	On <- A	2
OUTP r	O(C) <- r	2
OUTI	O(C) <- (HL) B <- B - 1 HL <- HL + 1	2
OUTIR	repeat OUTI until B=0	2
OUTD	O(C) <- (HL) B <- B - 1 HL <- HL - 1	2
OUTDR	repeat OUTD until B=0	2

Appendix B

Summary of Pseudo-Operation Mnemonics

.ASCII svalue | [n] ...

The .ASCII pseudo-op enters 7-bit ASCII characters into the program. The text is either entered as a string value, or as a numeric value enclosed in square brackets ([]), and the two forms may be intermixed and repeated as desired (with optional seperating commas).

.ASCIS svalue | [n] ...

The .ASCIS pseudo-op enters 7-bit ASCII characters into the program, and flags the last character by setting its high-order bit on. The format of the svalue is the same as for the .ASCII pseudo-op.

.ASCIZ svalue | [n] ...

The .ASCIZ pseudo-op enters 7-bit ASCII characters into the program, and flags the end of the characters by inserting an additional null byte. The format of the svalue is the same as for the .ASCII pseudo-op.

.BLKB nn

The .BLKB pseudo-op reserves a block of contiguous storage nn bytes long.

.BLKW nn

The .BLKW pseudo-op reserves a block of contiguous storage nn words long (nn x 2 bytes).

.BYTE {[r]}n {, {[r]}n ...}

The .BYTE pseudo-op enters single byte values into the program. Multiple values may be entered by separating them with a comma, and a single value may be preceeded by an optional repeat count.

.DATE

The .DATE pseudo-op generates an eight byte constant containing the current date in the format: mm/dd/yy.

.DEFINE symbol[arg1,arg2,...]=[text]

The .DEFINE pseudo-op defines a macro with the name symbol. arg1 through argn are optional dummy arguments. The body of the macro is represented by text.

.DEFINE symbol=[text]

The .DEFINE pseudo-op may also be used to define string variables with values equal to the specified text.

.END nn

The .END pseudo-op signals the end of the assembly. When encountered during pass 1, it initiates pass 2. During a listing pass, it initiates the listing of the symbol table (if not previously suppressed by the .XSYM pseudo-op). During an object pass, it generates an EOF record in the object file containing the value nn as the starting address of the object program.

.ENTRY symbol1 {, symbol2 ...}

The .ENTRY pseudo-op identifies the internally defined symbols which are subroutine library entry points to this program. Multiple symbols may be identified by separating them with commas.

.ERROR svalue

The .ERROR pseudo-op causes an "*" error to occur, forcing the listing of the current line, and an error notification. The svalue is treated as a .REMARK.

.EXIT

The .EXIT pseudo-op causes an immediate exit from the current macro expansion to occur.

.EXTERN symbol1 {, symbol2 ...}

The .EXTERN pseudo-op defines those symbols which are referenced in this program but are defined in another, separately assembled, program. Multiple symbols can be defined by separating them with commas.

.GOTO mlabel

The .GOTO pseudo-op is used within a macro to branch to the specified macro label.

.I8080

The .I8080 pseudo-op enables the Z warning message. This warning will be given whenever a machine operation unique to the Z80 is encountered.

.IDENT symbol

The .IDENT pseudo-op gives the module a name for later use by the linkage editor.

.INSERT {d:} file {.ext}

The .INSERT pseudo-op causes the specified PDOS file to be copied into the assembly in place of the .INSERT.

.INTERN symbol1 {, symbol2 ...}

The .INTERN pseudo-op identifies those symbols which are defined in this program and which will be referenced as external symbols by some separately assembled program. Multiple symbols may be identified by separating them with commas.

.LADDR

The .LADDR pseudo-op change the listing mode from displaying 16-bit quantities to displaying the Z80 image with the least significant byte first.

.LALL

The .LALL pseudo-op causes the assembler to list every text character processed, including those suppressed in the normal listing.

.LCTL

The .LCTL pseudo-op causes the assembler to list all listing control statements.

.LINK

The .LINK pseudo-op causes the assembler to output linkage information to the object file.

.LIST

The .LIST pseudo-op resumes a listing which has been stopped by the .XLIST pseudo-op.

.LIMAGE

The .LIMAGE pseudo-op changes the listing mode to display every byte of object code generated rather than the normal mode of a maximum of six bytes per statement.

.LOC nn

The .LOC pseudo-op changes the value of the assembler's program counter to nn. If nn is relocatable, then all labels will be assigned relocatable values. If it is absolute, then absolute values will be assigned.

.LSYM

The .LSYM pseudo-op reenables the listing of the symbol table during the .END pseudo-op processing after it has been disabled by the .XSYM pseudo-op. The .LSYM pseudo-op must occur prior to the .END pseudo-op to be effective.

.MASYN symbol1,symbol2

The .MASYN pseudo-op allows the definition of a new macro to be the same as a previously defined one. Symbol2 is defined to be a macro identical to the one defined as symbol1.

.OPSYN symbol1,symbol2

The .OPSYN pseudo-op allows the definition of a new op code mnemonic as a synonym of an already existing one. The symbol1 must be a defined machine or pseudo op code (or one previously defined using .OPSYN), symbol2 will be defined to be the same operation.

.PABS

The .PABS pseudo-op signals that the object file produced from this point in the assembly on is to be in absolute (INTEL compatable) format.

.PAGE

The .PAGE pseudo-op causes a skip to the top of the next page during a listing pass.

.PAGE w {, l}

The .PAGE pseudo-op is used in this manner to specify the page size for the assembly listing. The width, and optionally the length, may be given.

.PBIN

The .PBIN pseudo-op specifies that the object file is to be produced in binary.

.PHEX

The .PHEX pseudo-op specifies that the object file is to be produced in ASCII.

.PREL

The .PREL pseudo-op signals that the object file produced from this point in the assembly on is to be in relocatable (TDL relocatable) format.

.PRGEND

The .PRGEND pseudo-op is used in place of the .END pseudo-op to terminate each module in a library file assembly. After pass 2 through the module terminated by .PRGEND, the next module following the .PRGEND will be assembled.

.PRNTX svalue

The .PRNTX pseudo-op will cause its string value to be printed on the console whenever it is encountered in the assembly process.

.PROGID id,ver,rev

The .PROGID pseudo-op provides program id, version and revision for output in the object file, and for use by the linkage editor in identifying the program.

.PSYM

The .PSYM pseudo-op signals that the entire symbol table from the assembly is to be output at the end of the object file. The .PSYM pseudo-op must appear prior to the .END pseudo-op to be effective.

.RADIX n

The .RADIX pseudo-op changes the default base in which a numeric constant is interpreted during the assembly to n. The valid values for n are 2, 8, 10, or 16. The value is always interpreted as a decimal number.

.RAD40 symbol

The .RAD40 pseudo-op generates a unique 4 byte value in radix-40 notation for the symbol given. The symbol must conform to the rules for any symbol in the assembly. This pseudo-op is used mostly for developing system software utilizing symbol tables.

.RELOC

The .RELOC pseudo-op restores the value of the assembler's program counter to whatever it was before the immediately preceding .LOC pseudo-op.

.REMARK svalue

The .REMARK pseudo-op allow the entry of multiple line comments into the source program.

.RLIST

The .RLIST pseudo-op restores the listing control flags from the top element of the .SLIST push-down stack.

.SALL

The .SALL pseudo-op suppresses all macro expansions on the assembly listing (normally all lines generating code are listed).

.SBTTL svalue

The .SBTTL pseudo-op sets the sub-title for the assembly listing to the specified string value (which must be less than 72 characters in length). If the .SBTTL pseudo-op is the first operation after a .PAGE, the sub-title will appear on the new page.

.SLIST

The .SLIST pseudo-op saves the current listing control flags on the top of a four element push-down stack.

.SYN symbol1,symbol2

The .SYN pseudo-op makes any two symbols synonymous. The symbol tables are searched for symbol1 in the normal operand field order (label/symbol, macro, opcode), and symbol2 is defined to have the same value as symbol1.

.SYSYN symbol1,symbol2

The .SYSYN pseudo-op makes one symbol the synonym of an already defined symbol/label. The value of a symbol/label symbol1 is obtained, and symbol2 is defined to be the same type and value.

.TEMPS nn

The .TEMPS pseudo-op allocates temporary array space for global and local variables.

.TIME

The .TIME pseudo-op generates an eight byte constant containing the current time in the format hh:mm:ss.

.TITLE svalue

The .TITLE pseudo-op sets the title for the assembly listing to the specified string value (which must be less than 72 characters in length). The title is put at the top of every page during a listing. If the .TITLE pseudo-op is the first operation after a .PAGE pseudo-op, the title will be listed on the new page.

.WORD nn {, nn ...}

The .WORD pseudo-op enters 2-byte values into the program in proper Z80 format (least significant byte first). Multiple values may by entered by separating them with a comma.

.XADDR

The .XADDR pseudo-op is used after a .LADDR pseudo-op to return to the standard format of listing 16-bit values.

.XALL

The .XALL pseudo-op is used after a .LALL or .SALL pseudo-op to return to the standard listing mode.

.XCTL

The .XCTL pseuod-op is used after a .LCTL pseudo-op to return the the standard mode of suppressing the listing of listing control statements.

.XIMAGE

The .XIMAGE pseudo-op is used after a .LIMAGE pseudo-op to return to the standard listing mode of only six object bytes per statement.

.XLINK

The .XLINK pseudo-op is used after a .LINK pseudo-op to suppress the inclusion of linkage information in the object file.

.XLIST

The .XLIST pseudo-op suppresses the listing of all following statements (until a .LIST pseudo-op is encountered).

.XPSYM

The .XPSYM pseudo-op disables the output of the symbol table at the end of the object file after it has been enabled by the .PSYM pseudo-op. The .XPSYM pseudo-op must occur prior to the .END pseudo-op to be effective.

.XSYM

The .XSYM pseudo-op disables the listing of the symbol table by the .END pseudo-op (unless reenabled by the .LSYM pseudo-op). The .XSYM pseudo-op must appear before the .END pseudo-op to be effective.

.Z80

The .Z80 pseudo-op is used to disable the effect of a previous .I8080 pseudo-op. This inhibits the Z warning message on machine operations unique to the Z80.

.IFx arg,[true text] ... {[false text]}

The .IFx pseudo-op will assemble the true text specified only if the particular condition being tested for is true, The optional false text is assembled if the condition is false. The .IFx pseudo-ops and their conditions are as follows:

.IF1: assembling pass 1
.IF2: not assembling pass 1
.IFB: blank
.IFDEF: defined
.IFDIF: different
.IFE: zero or blank
.IFG: positive
.IFGE: zero or positive
.IFIDN: identical
.IFL: negative
.IFLE: zero or negative
.IFN: not zero
.IFNB: not blank
.IFNDEF: not defined

Appendix C

Error Codes

Errors in the source program encountered during the assembly process are indicated on the listing by a single letter code at the left of the statement in error. Although the assembler may detect more than two errors per statement, only the first two codes are given. As an added aid to locating the error in the statement, a question mark is printed to the right of the character which triggered the error. All errors generate a question mark, even if they are not one of the first two per statement.

The following is a list of the error codes and their meanings:

A Argument error. This is a broad class of errors which may be caused by many different things.

B Bad macro error. Either an error in a macro definition or a call on a bad macro.

D Duplicate symbol reference error. The symbol flagged is multiply-defined. The first value given to the symbol is used in the assembly.

E External symbol error. An external symbol is improperly used in the statement.

I Internal symbol error. An internal symbol is improperly used in the statement.

L Label error. An invalid character has been found in the label field of the statement.

M Multiply-defined symbol error. A symbol is defined more than once. This error is given mostly during Pass 1. During the other passes, it usually will appear as a phase error (P).

O Operation error. The symbol in the operation field is not a valid machine operation code, macro name, or symbol.

P Phase error. A label is assigned a value during Pass 2 (or 3 or 4) which is different than that assigned during Pass 1.

Q Questionable error. This is a broad class of warnings which the assembler gives when it finds ambiguous statements. Q errors may or may not generate correct code. The assembler will attempt to do what the programmer intended.

R Relocation error. A relocatable symbol or expression is incorrectly used (eg. in a .BLKB pseudo-op).

S Subscript error. A subscript has exceeded the current temporary array space allocation.

T Table overflow. One of the Assembler's internal tables has overflowed. The Assembler will attempt to continue, but no new labels or macros will be defined.

U Undefined label/symbol error. A symbolic reference which was never defined is used in the statement.

X Index error. Another character appears in a statement at a point where only an index register reference is allowed (X or Y).

Z Z80 error. A Z80 machine operation has been encountered while in 8080 mode (.I8080). This is only a warning and the opcode will be properly assembled.

* User defined macro error. A .ERROR pseudo-op was encountered.

Appendix D

Object File Formats

The PSA Assembler produces two different object file formats depending on the use of the .PABS and the .PREL pseudo-ops. It also outputs the two formats two different ways, binary (.PBIN) and ASCII (.PHEX). Each of the two formats will be described seperately, and where differences between binary and ASCII exist, they will be noted. In addition, the .XLINK option allows the suppression of some of the information in the relocatable format to allow the direct production of a relocatable core image module instead of a relocatable object module.

TDL Object Module Format Definition

The use of the .PREL pseudo-op (which is default if neither is specified) causes the generation of the TDL Object Module Format (Copyright 1976 by Technical Design Labs, Inc.). This format allows for simple relocation of complete programs by the PSA Debugging System (BUG), and for complex relocation and linking of modules by the PSA Linkage Editor (LINK).

The TDL Object Module Format consists of a sequential file of ASCII characters representing the binary data, symbol, and control information required to construct a final program from the module. All binary bytes within this structure are represented as two ASCII characters corresponding to the hexidecimal value of the byte (eg. 11001001 -> C9). All ASCII values are represented by the corresponding ASCII character (eg. A -> A). In the binary output mode, the format is basically the same, but all binary bytes are represented by themselves, not as two ASCII characters.

Each of the different records within the module is indicated by the use of a prompt character as the first character of the record. The valid prompt characters are:

```
! -> module identification record
+ -> program identification record
@ -> entry point record
# -> internal symbol record
\ -> external symbol/relocation base record
& -> symbol table record
; -> data/program/end-of-file record
```

(Note that only the records prompted by a ; are output if the .XLINK mode is in effect.)

Every record in the module is terminated by a one byte binary checksum of all of the preceeding bytes in the record except for the prompt character. The checksum is the two's complement of the sum of the preceeding bytes. Any output format (two character binary, one character ASCII or one byte binary) still counts as only one byte in the checksum (ie. before conversion for output).

In addition, each record in the ASCII output mode is preceded by a carriage return/line feed sequence to facilitate listing the module on an external device. It is not present in the binary output mode.

The following descriptions are specified assuming ASCII output mode. With the above noted exception of the carriage return/line feed preceding each record, the binary format is identical, with each binary byte being left unexpanded. ASCII characters are left as they are in either mode.

Module Identification Record (!)

Byte	1-2	CR/LF
	3	Exclamation point (!) prompt.
	4-9	ASCII module name.
	10-11	Checksum.

Program Identification Record (+)

Byte	1-2	CR/LF
	3	Plus (+) prompt.
	4-9	ASCII program identification.
	10-11	Version number (in hex).
	12-13	Revision number (in hex).
	14-19	Date of assembly (ASCII MMDDYY) if available.
	20-25	Time of assembly (ASCII HHMMSS) if available.
	26-27	Checksum.

Entry Point Record (@)

Byte	1-2	CR/LF
	3	At-sign (@) prompt.
	4-5	Number of entry points in this record.
	6-??	ASCII names of entry points, 6 bytes per name. The names are left justified and blank filled.
	??	Checksum.

Internal Symbol Record (#)

Byte 1-2	CR/LF
3	Pound sign (#) prompt.
4-5	Number of internal symbols in this record.
6-11	ASCII name of internal symbol, left justified and blank filled.
12-13	Relocation base for symbol. The value of this symbol is relative to the relocation base specified.
14-17	Symbol value (16 bit).
....	The above three fields are repeated for each internal symbol in the record.
??	Checksum.

External Symbol/Relocation Base Record (\)

Byte 1-2	CR/LF
3	Back-slash (\) prompt.
4-5	Number of external/relocation symbols in this record.
6-11	ASCII name of the symbol, left justified and blank filled.
12-13	Relocation number assigned to this symbol in this module. This number is unique for each symbol. It starts with one and increases sequentially for each subsequent external/relocation base symbol.
14-17	Relocation segment size/external reference flag. If this value is zero, it represents a reference to a symbol defined externally to this module (usually a subroutine or global data item). If it is non-zero, then the value is the size of the relocation segment as defined in this object module. This segment can contain either code or data, and may be located anywhere in memory by the linkage editor, independent of any other segment.
....	The above three fields are repeated for each symbol contained in this record.
??	Checksum.

Symbol Table Record (&)

Byte 1-2	CR/LF
3	Ampersand (&) prompt.
4-??	The remainder of this record is identical to the internal symbol record. All symbols defined in this module are contained in these records.

Data/Program Record (;)

Byte 1-2 CR/LF

3 Semicolon (;) prompt

4-5 Number of binary data bytes in this record. The maximum is 32 binary bytes (64 bytes of ASCII representation). If this value is zero, this record is a end-of-file record, described below.

6-9 Load address of the data relative to the specified relocation base.

10-11 Relocation base for all relocation in this record. All relocatable values in this record are added to the current value of the specified relocation base before being put into memory. (If .XLINK is in effect, the only allowable relocation bases are 0 and 1.)

12-13 Relocation control byte. This byte controls the relocation of the next eight bytes in the record (if that many remain according to the count field). The bits are used from left to right. The bits have the following meanings:

- 0: a single absolute byte -> load unmodified.
- 10: a two byte relocatable value, least significant byte first -> add the 16 bit value to the current relocation base, and load the result least significant byte first. (If .XLINK is in effect, and the current relocation base is 0, then the 16 bit value is added to relocation base 1.)
- 110: a three byte reference to a different relocation base. The first byte is the relocation base number, and the two after that are the 16 bit value, least significant byte first -> add the specified relocation base to the 16 bit value, and load the result least significant byte first. (In .XLINK mode, this control pattern is not generated.)
- 111: a two byte reference to a different relocation base. The first byte is the relocation base number, and the following byte is the 8 bit value -> add the specified relocation base to the 8 bit value, and load the byte. If the result is not between -128 and +255, give an error.

Note that a two or three byte combination is never broken across a record boundary.

14-29 Data bytes controlled as above.

30-?? The above control/data byte combinations are repeated as specified by the count.

?? Checksum.

End-of-File Record (;)

Byte 1-2 CR/LF

3 Semicolon (;) prompt.

4-5	Zero to indicate end-of-file record.
6-9	Starting address for module relative to the specified relocation base. This address is optionally generated by the language processor, and may be zero.
10-11	Relocation base for starting address. (In .XLINK mode may be only 0 or 1.)
12-13	Checksum.

INTEL Object Format

The use of the .PABS pseudo-op causes an INTEL "hex" object module to be produced. This object format can also be loaded by the PSA Debugging System, but provides no relocatability.

All of the above comments concerning byte formats and checksums apply to this format as well.

Byte 1-2	CR/LF
3	Colon (:) prompt.
4-5	Number of binary data bytes in this record. The maximum number is 32 binary bytes (64 bytes of ASCII representation). If this value is zero, this record is an end-of-file record, and the load address is the program starting address.
6-9	Load address of the data in this record.
10-11	Unused.
12-??	Data bytes.
??	Checksum.

Appendix E

Assembler Operation under PDOS

The PSA Macro Assembler is initiated by the PDOS command:

PASM {sd:}file{.ext}{[user]} {dd:}{switches}

where

- sd is the optional PDOS disk specification for the source file (defaults to the logged in disk)
- file is the source file name
- ext is the optional source file extension (defaults to ASM)
- user is the optional source file user number (defaults to logged-in user)
- dd is the optional PDOS disk specification for the output files (defaults to the same as the source file)
- switches are the optional assembly control switches, each of which is a single letter and which may appear in any order (with no intervening spaces)

The object file created by the assembly will have the same name as the source file, with an extension of .HEX if the .PABS option was used, and .REL if the .PREL option was used (the default).

Switches

A .LALL
B listing to both disk and list device
C .LCTL
D listing to disk (file name same as source with extension of PRN)
H .PHEX (default is .PBIN)
I .LIMAGE
K .XLINK (default is .LINK)
L listing only - no object file generated
O object only - no listing generated
P .PSYM
S .SALL
X .XLIST
Y .XSYM

Note that all switches with pseudo-op equivalents will be overidden by contrary pseudo-ops within the source program.

Assembly Time Control

While an assembly is taking place, a number of console control options are available. A control-C will abort the assembly back to the operating system with a return code of 255. A control-S will temporarily halt the assembly, after which a control-C will abort it, or a control-Q will resume it. A control-T will temporarily halt the assembly at the top of the next output page of the listing. When control-T is entered on the console, nothing will happen until the next top-of-page is reached, at which time the assembler will act as if a control-S had been entered (see above).

The assembler assumes that the paper is positioned at its top print line prior to the start of the listing. The assembler will then count output lines and put a page number and heading at the top of every page. The page is assumed to be 79 columns wide and 66 lines long unless changed by the .PAGE pseudo-op. A two line margin is always left at the top and bottom of the page.

LIFEBOAT ASSOCIATES SOFTWARE PROBLEM REPORT

Please use this form to report errors or problems in software supplied by Lifeboat Associates. This form is designed to act as a transmittal sheet.

Software Product Name: ____________________ Media Format: ________

Version No.: ________ Serial No.: ________ Invoice No.: ________

Purchased From: ____________________

____

____

Date of Purchase: ____________ Return Authorization #: ________
Has the software registration card been returned? ______

Computer Used: ____________ CPU (8080/8085/Z-80): ________

Disk Capacity: ________ Number of Drives: ____ Memory Size: ________

Operating System/Version (If not listed above): ________/________

Software used with the above product, (e.g. list the BASIC used if you are reporting a problem with a Payroll program that uses it).

Name of Software	Version
____________________	________
____________________	________

Does the software come with sample or test programs? ______
If so, have you been able to use them successfully? ______

Please describe the problem you have encountered. Include references to the manual if appropriate. Try to reduce the problem to a simple test case. Enclose any appropriate programs (preferably on disk). If you feel that the problem may be caused by the disk being defective, you may prefer to return the original disk with this report to achieve the fastest resolution of the problem. (If so, call for a Return Authorization No. A handling charge may be incurred. No handling charge will be made if a product or portion thereof is returned **DUE TO DISKETTE MEDIA DEFECTS** within 30 days from the date of sale).

Information on product changes, bugs, fixes and current version numbers are published in **Lifelines**, our software newsletter.

PROBLEM DESCRIPTION: (Continue on additional pages if necessary)

Area Phone Num. Ext.

Name: ____________________ (____) ___-_____ (____)

Address: ____________________ (____) ___-_____ (____)

City: ____________ State: ________ Zip Code: ________

Return to: Lifeboat Associates

Technical assistance is available